Global Inverse Consistency for Interactive Constraint Satisfaction*

Christian Bessiere[1], Hélène Fargier[2], and Christophe Lecoutre[3]

[1] LIRMM-CNRS, University of Montpellier, France
[2] IRIT-CNRS, University of Toulouse, France
[3] CRIL-CNRS, University of Artois, Lens, France
bessiere@lirmm.fr, fargier@irit.fr, lecoutre@cril.fr

Abstract. Some applications require the interactive resolution of a constraint problem by a human user. In such cases, it is highly desirable that the person who interactively solves the problem is not given the choice to select values that do not lead to solutions. We call this property *global inverse consistency*. Existing systems simulate this either by maintaining arc consistency after each assignment performed by the user or by compiling offline the problem as a multi-valued decision diagram. In this paper, we define several questions related to global inverse consistency and analyse their complexity. Despite their theoretical intractability, we propose several algorithms for enforcing global inverse consistency and we show that the best version is efficient enough to be used in an interactive setting on several configuration and design problems. We finally extend our contribution to the inverse consistency of tuples.

1 Introduction

Constraint Programming (CP) is widely used to express and solve combinatorial problems. Once the problem is modelled as a constraint network, efficient solving techniques generate a solution satisfying the constraints, if such a solution exists. However, there are situations where the user has strong opinions about the way to build good solutions to the problem but some of the desirable/undesirable combinations will become clear only once some of the variables are assigned. In this case, the constraint solver should be there to assist the user in the solution design and to ensure her choices remain in the feasible space, removing the combinatorial complexity from her shoulders. See the Synthia system for protein design as an early example of using CP to interactively solve a problem [12]. Another well known example of such an interactive solving of constraint-based models is product configuration [7, 1]. The person modelling the product as a constraint network for the company knows its technical and marketing requirements. She models the feasibility, availability and/or marketing constraints about the product. This constraint network captures the catalog of possible products, which may contain billions of solutions, but in an intentional and compact way. Nevertheless, the modeller does not know the constraints or preferences of the customer(s). Now, this

* This work has been funded by the ANR ("Agence Nationale de la Recherche") project BR4CP (ANR-11-BS02-008).

is the customer who will look for solutions, with her own constraints and preferences on the price, the colour, or any other configurable feature.

These applications refer to an interactive solving process where the user selects values for variables according to her own preferences and the system checks the constraints of the network, until all variables are assigned and satisfy all constraints of the network. This solving policy raises an important issue: the person who interactively solves the problem should not be led to a dead-end where satisfying all constraints of the network is impossible. Existing interactive solving systems address this issue either by compiling the constraint network into a multivalued decision diagram (MDD) at the modelling phase [1, 9, 10] or by enforcing arc consistency on the network after each assignment performed by the user [12]. Compiling the constraint network as a MDD can require a significant amount of time and space. That is why compilation is performed offline (before the solving session). As a consequence, configurators based on a MDD compilation are restricted to static constraint networks: non-unary constraints can neither be added nor removed once the network compiled. It is thus not possible for the user to perform complex requirements, e.g., she is interested in travelling to Venezia only during the carnival period. Arc and dynamic arc consistencies require a lighter computational effort but the user can be trapped in dead-ends, which is very risky from a commercial point of view. It has been shown in [5] that arc consistency (and even higher levels of local consistency) can be very bad approximations of the ideal state where all values remaining in the network can be extended to solutions.

The message of our paper is that for many of the problems that require interactive solving of the problem, and especially for real problems, it is computationally feasible to maintain the domains of the variables in a state where they only contain those values which belong to a complete solution extending the current choices of the user. Inspired by the nomenclature used in [6] and [15], we call this level of consistency *global inverse consistency* (GIC).

Our contribution addresses several aspects. First, we formally characterise the questions that underlie the interactive constraint solving loop and we show that they are all NP-hard. Second, we provide several algorithms with increasing sophistication to address those tasks and we experimentally show that the most efficient one is efficient enough to be used in an interactive constraint solving loop of several non trivial configuration and design problems. Third, we finally extend all these contributions to the *positive consistency* of constraints, which is a problem closely related to GIC that appears in configuration.

2 Background

A (discrete) constraint network (CN) N is composed of a finite set of n variables, denoted by $vars(N)$, and a finite set of e constraints, denoted by $cons(N)$. Each variable x has a domain which is the finite set of values that can be assigned to x. The initial domain of a variable x is denoted by $dom^{init}(x)$ whereas the current domain of x is denoted by $dom(x)$; we always have $dom(x) \subseteq dom^{init}(x)$. The maximum domain size for a given CN will be denoted by d. To simplify, a variable-value pair (x, a) such that $x \in vars(N)$ and $a \in dom(x)$ is called a value of N. Each constraint c involves

an ordered set of variables, called the *scope* of c and denoted by $scp(c)$, and is semantically defined by a relation, denoted by $rel(c)$, which contains the set of tuples allowed for the variables involved in c. The *arity* of a constraint c is the size of $scp(c)$, and will usually be denoted by r.

An *instantiation* I of a set $X = \{x_1, \ldots, x_k\}$ of variables is a set $\{(x_1, a_1), \ldots, (x_k, a_k)\}$ such that $\forall i \in 1..k, a_i \in dom^{init}(x_i)$; X is denoted by $vars(I)$ and each a_i is denoted by $I[x_i]$. An instantiation I *on* a CN N is an instantiation of a set $X \subseteq vars(N)$; it is *complete* if $vars(I) = vars(N)$. I is *valid* on N iff $\forall (x, a) \in I, a \in dom(x)$. I *covers* a constraint c iff $scp(c) \subseteq vars(I)$, and I *satisfies* a constraint c with $scp(c) = \{x_1, \ldots, x_r\}$ iff (i) I covers c and (ii) the tuple $(I[x_1], \ldots, I[x_r]) \in rel(c)$. An instantiation I on a CN N is *locally consistent* iff (i) I is valid on N and (ii) every constraint of N covered by I is satisfied by I. A *solution* of N is a complete locally consistent instantiation on N; $sols(N)$ denotes the set of solutions of N. A CN N is *satisfiable* iff $sols(N) \neq \emptyset$.

The ubiquitous example of constraint propagation is enforcement of *generalised arc consistency* (GAC) which removes values from domains without reducing the set of solutions of the constraint network. A value (x, a) of a CN N is GAC on N iff for every constraint c of N involving x, there exists a valid instantiation I of $scp(c)$ such that I satisfies c and $I[x] = a$. N is GAC iff every value of N is GAC. Enforcing GAC means removing GAC-inconsistent values from domains until the constraint network is GAC. In this paper, we shall refer to MAC which is an algorithm considered to be among the most efficient generic approaches for the solution of CNs. MAC [17] explores the search space depth-first and enforces (generalised) arc consistency after each decision taken (variable assignment or value refutation) during search. A *past* variable is a variable explicitly assigned by the search algorithm whereas a *future* variable is a variable not (explicitly) assigned. The set of future variables of a CN N is denoted by $vars^{fut}(N)$.

3 Problems Raised by Interactive Constraint Solving

In this section we formally characterise the questions that underlie the interactive constraint solving loop and we study their theoretical complexity.

3.1 Formalization

We first need to define global inverse consistency.

Definition 1 (Global Inverse Consistency). *A value (x, a) of a CN N is* globally inverse consistent (GIC) *iff $\exists I \in sols(N) \mid I[x] = a$. A CN N is GIC iff every value of N is GIC.*

The *GIC closure* of N is the CN obtained from N by removing all the values that do not belong to a solution of N. The obvious problems that follow are to check whether a constraint network is GIC or not, and to enforce GIC.

Problem 1 (Deciding GIC) *Given a CN N, is N GIC?*

Problem 2 (Computing GIC) *Given a CN N, compute the GIC closure of N.*

As we are interested in interactive solving, we define the problem of restoring (maintaining) GIC after the user has performed a variable assignment.

Problem 3 (Restoring GIC) *Given a CN N that is GIC, and a value (x, a) of N, restore GIC after the assignment $x = a$ has been performed.*

In a configuration setting, as soon as some mandatory variables have been set, the user can ask for an automatic completion of the remaining variables. Hence the definition of following problem:

Problem 4 (Solving a GIC network) *Given a CN N that is GIC, find a solution to N.*

3.2 Complexity Results

Not surprisingly, the basic questions related to GIC (Problems 1 and 2) are intractable.

Theorem 1 (Problem 1). *Deciding whether a constraint network N is GIC is NP-complete, even if N is satisfiable.*

Proof. We first prove membership to NP. For each value (x, a) of N, it is sufficient to provide a solution I of N such that the projection $I[x]$ of I on variable x is equal to a. This certificate has size $n \cdot n \cdot d$ and can be checked in polynomial time.

Completeness for NP is proved by reducing 3COL to the problem of deciding whether a satisfiable CN is GIC. Take any instance of the 3COL problem, that is, a graph $G = (V, E)$. Consider the CN N where $vars(N) = \{x_i \mid i \in V\}$, $dom(x_i) = \{0, 1, 2, 3\}, \forall i \in V$, and $cons(N) = \{(x_i \neq x_j) \vee (x_i = 0 \wedge x_j = 0) \mid (i, j) \in E\}$. Clearly $[0, \ldots, 0]$ is a solution of N, and by construction, N has other solutions iff G is 3-colourable. Now, if G is 3-colourable, N is GIC because colours are completely interchangeable. Therefore, N is GIC iff G is 3-colourable. □

Our proof shows that hardness for deciding GIC holds for binary CNs (i.e., CNs only involving binary constraints). We have another proof, inspired from that used in Theorem 3 in [2], that shows that deciding GIC is still hard for Boolean domains and quaternary constraints.

Theorem 2 (Problem 2). *Computing the GIC closure of a constraint network N is NP-hard and NP-easy, even if N is satisfiable.*

Proof. We prove NP-easiness by showing that a polynomial number of calls to a NP oracle are sufficient to build the GIC closure of N. For each value (x, a) of N, we ask the NP oracle whether N with the extra constraint $x = a$ is satisfiable (we call this an *inverse check*). Once all values have been tested, we build the GIC closure of N by removing from each $dom(x)$ all values a for which the oracle test returned 'no'. Hardness is a direct corollary of Theorem 1. □

Notice that the two previous intractability results are still valid when the CN is satisfiable, as is the case at the beginning of an interactive resolution session.

We finally show that Problems 3 and 4 are unfortunately not easier than checking GIC or enforcing GIC from scratch. But they are not harder.

Theorem 3 (Problem 3). *Given a CN N that is GIC, and a value (y, b) of N, computing the GIC closure of the CN N', where $vars(N') = vars(N)$ and $cons(N') = cons(N) \cup \{y = b\}$ is NP-hard and NP-easy.*

Proof. NP-easiness is proved as in the proof of Theorem 2 by showing that a polynomial number of calls to a NP oracle are sufficient to build the GIC closure of N'. For each value (x, a) of N (except values (y, a) with $a \neq b$), we ask the NP oracle whether N' with the extra constraint $x = a$ is satisfiable. Once all values have been tested, we build the closure of N' by removing from $dom(y)$ all values $a \neq b$ and removing from each $dom(x)$ all values a for which the oracle test returned 'no'. Hardness is a direct corollary of Theorem 7 in [2]. □

Theorem 4 (Problem 4). *Generating a solution to a GIC constraint network cannot be done in polynomial time, unless $P = NP$.*

Proof. The following proof is derived from [16]. But it is also a corollary of the recent and more complex Theorem 3.1 in [8].

Suppose we have an algorithm A that generates a solution to a GIC constraint network N in time bounded by a polynom $p(|N|)$. Take any instance of the 3COL problem, that is, a graph $G = (V, E)$. Consider the CN N where $vars(N) = \{x_i \mid i \in V\}$, $dom(x_i) = \{0, 1, 2\}, \forall i \in V$, and $cons(N) = \{x_i \neq x_j \mid (i, j) \in E\}$. N has a solution iff G is 3-colourable. Now, if G is 3-colourable, N is GIC because colours are completely interchangeable. Thus, it is sufficient to run A during $p(|N|)$ steps. If it returns a solution to N, then the 3COL instance is satisfiable. Otherwise, the 3COL instance is unsatisfiable. Therefore, as 3COL is NP-complete, there cannot exist a polynomial algorithm for generating a solution to a GIC constraint network, unless $P = NP$. □

4 GIC Algorithms

In this section, we introduce four algorithms to enforce global inverse consistency. These GIC algorithms use increasingly sophisticated data structures and techniques that have recently proved their worth in filtering algorithms proposed in the literature; e.g., see [14, 18]. To simplify our presentation, we assume that the CNs are satisfiable, which is the case in interactive resolution, allowing us to avoid handling domain wipe-outs in the GIC procedures. Note that these algorithms can be used to enforce GIC, but also to maintain it during a user-driven search. This is why we refer to the set $vars^{fut}(N)$ of future variables in some instructions.

The first algorithm, GIC1, described in Algorithm 1, is really basic: it will be used as our baseline during our experiments. For each value a in the domain of a future variable x, a solution for the CN N where x is assigned the value a, denoted by $N|_{x=a}$, is sought using a complete search algorithm. This search algorithm, called here searchSolutionFor, either returns the first solution that can be found, or the special value nil. Our implementation choice will be the algorithm MAC that maintains (G)AC during a backtrack search [17]. Hence, in Algorithm 1, when it is proved with searchSolutionFor that no solution exists, i.e., $I = nil$, the value a can be deleted. Note that, in contrary to

Algorithm 1: GIC1(N: CN)

```
1  foreach variable x ∈ vars^fut(N) do
2      foreach value a ∈ dom(x) do
3          I ← searchSolutionFor(N|_{x=a})
4          if I = nil then
5              remove a from dom(x)
```

Algorithm 2: handleSolution2/3(x: variable, I: instantiation)

```
1  foreach variable y ∈ vars^fut(N) | y is revised after x do
2      if stamp[y][I[y]] ≠ time then
3          stamp[y][I[y]] ← time
4          nbGic[y] + +
```

Algorithm 3: isValid(X : set of variables, I : instantiation): Boolean

```
1  foreach variable x ∈ X do
2      if I[x] ∉ dom(x) then
3          return false
4  return true
```

Algorithm 4: GIC2/3(N: CN)

Data: GIC3 is obtained by considering light grey coloured instructions between lines 5 and 6, and after line 10

```
1  time + +
2  foreach variable x ∈ vars^fut(N) do
3      nbGic[x] ← 0
4  foreach variable x ∈ vars^fut(N) | nbGic[x] < |dom(x)| do
5      foreach value a ∈ dom(x) | stamp[x][a] < time do
           if isValid(vars(N),residue[x][a]) then
               handleSolution2/3(x,residue[x][a])
               continue
6          I ← searchSolutionFor(N|_{x=a})
7          if I = nil then
8              remove a from dom(x)
9          else
10             handleSolution2/3(x,I)
               residue[x][a] ← I
```

weaker forms of consistency, when a value is pruned there is no need for GIC to repeat the process of iterating over the values remaining in the CN.

The second algorithm, GIC2 described in Algorithm 4 (ignoring light grey lines), uses timestamping. This is useful when GIC is maintained during a user-driven search. We use an integer variable `time` for counting time, and we introduce a two-dimensional array `stamp` that associates with each value (x, a) of the CN the last time (value of $\texttt{stamp}[x][a]$) a solution was found for that value. We also assume that variables are implicitly totally ordered (for example, in lexicographic order). Then, the idea is to increment the value of the variable `time` whenever a new call to GIC2 is performed (see line 1) and to test `time` against each value (x, a) of the CN (see line 5) to determine whether it is necessary or not to search for a solution for (x, a). When a solution I is found, function handleSolution2/3 is called at line 10 in order to update stamps. Actually, we only update the stamps of values in I corresponding to variables that are processed after x in the loop of revisions (line 4) in Algorithm 4. These are the variables that have not been processed yet by the loop at line 4 of Algorithm 4. Finally, by further introducing a one-dimensional array `nbGic` that associates with each variable x of the CN the number of values in $dom(x)$ that have been proved to be GIC, it is possible to avoid some iterations of loop 5; see initialization at lines 2-3, testing at line 4 and update at line 4 of Algorithm 2.

The third algorithm, GIC3, described in Algorithm 4 when considering light grey lines, can be seen as a refinement of GIC2 obtained by exploiting residues, which correspond to solutions that have been previously found. Here, we introduce a two-dimensional array `residue` that associates with each value (x, a) of the CN the last solution found for this value (potentially, during another call to GIC3). Because residual solutions may not be valid anymore, for each value (x, a) we need to test the validity of $\texttt{residue}[x][a]$ by calling the function isValid; see instructions between lines 5 and 6. If the residue is valid, we call handleSolution2/3 to update the other data structures, and we continue with the next value in the domain of x. A validity test, Algorithm 3, only checks that all values in a given complete instantiation are still present in the current domains. Of course, when a new solution is found, we record it as a residue; see instruction after line 10.

Our last algorithm, GIC4 described in Algorithm 6, is based on an original use of simple tabular reduction [18]. The principle is to record all solutions found during the enforcement of GIC in a table, so that an (adaptation of an) algorithm such as STR2 [13] can be applied. The current table is given by all elements of an array `solutions` at indices ranging from 1 to `nbSolutions`. As for STR2, we introduce two sets of variables called S^{val} and S^{sup}. The former allows us to limit validity control of solutions to the variables whose domains have changed recently (i.e., since the last execution of GIC4). This is made possible by reasoning from domain cardinalities, as performed at lines 3 and 26–27 with the array `lastSize`. The latter (S^{sup}) contains any future variable x for which at least one value is not in the array $\texttt{gicValues}[\texttt{x}]$, meaning that it has still to be proved GIC. Related details can be found in [13]. After the initialization of S^{val} and S^{sup} (lines 1–8), each instantiation $\texttt{solutions}[\texttt{i}]$ of the current table is processed (lines 11–16). If it remains valid (hence, a solution), we update structures `gicValues` and S^{sup} by calling the function handleSolution4. Otherwise, this instanti-

Algorithm 5: handleSolution4(I : instantiation)

```
1  foreach variable x ∈ S^sup do
2      if I[x] ∉ gicValues[x] then
3          gicValues[x] ← gicValues[x] ∪ {I[x]}
4          if |gicValues[x]| = |dom(x)| then
5              S^sup ← S^sup \ {x}
```

Algorithm 6: GIC4(N: CN)

```
   // Initialization of structures
1  S^val ← ∅
2  foreach variable x ∈ vars(N) do
3      if |dom(x)| ≠ lastSize[x] then
4          S^val ← S^val ∪ {x}
5  S^sup ← ∅
6  foreach variable x ∈ vars^fut(N) do
7      gicValues[x] ← ∅
8      S^sup ← S^sup ∪ {x}
   // The table of current solutions is traversed
9  i ← 1
10 while i ≤ nbSolutions do
11     if isValid(S^val,solutions[i]) then
12         handleSolution4(solutions[i])
13         i++
14     else
15         solutions[i] ← solutions[nbSolutions]
16         nbSolutions − −
   // Search for values not currently supported is performed
17 foreach variable x ∈ S^sup do
18     foreach value a ∈ dom(x) \ gicValues[x] do
19         I ← searchSolutionFor(N|_{x=a})
20         if I = nil then
21             remove a from dom(x)
22         else
23             nbSolutions++
24             solutions[nbSolutions] ← I
25             handleSolution4(I)
26 foreach variable x ∈ vars^fut(N) do
27     lastSize[x] ← |dom(x)|
```

ation is deleted by swapping it with the last one. The rest of the algorithm (lines 17–25) just tries to find a solution support for each value not present in `gicValues`. When a new solution is found, it is recorded in the current table (lines 23–24) and handleSolution4 is called (line 25).

Theorem 5. *Algorithms GIC1, GIC2, GIC3 and GIC4 enforce GIC.*

Proof. (sketch) This is immediate for GIC1. For GIC2 and GIC3, the use of timestamps and residues permits us to avoid useless inverse checks. For GIC4, the same arguments as those used for proving that STR2 enforces GAC hold. Simply, additional inverse checks are performed for values not collected (in `gicValues`) during the traversal of the current table. □

The worst-case space complexity (for the specific data structures) of GIC1 is $O(1)$. For GIC2 and GIC3, this is $O(nd)$ because `nbGic` is $O(n)$, `stamp` and `residue` are $O(nd)$. For GIC4, S^{val}, S^{sup} and `lastSize` are $O(n)$, `gicValues` is $O(nd)$, and the structure `solutions` is $O(n^2d)$. The time complexity of the GIC algorithms can be expressed in term of the number of calls to the (oracle) searchSolutionFor. For GIC1, this is $O(nd)$. For GIC2, in the *best-case*, only d calls are necessary, one call permitting to prove (through timestamping) that n values are GIC. For GIC3 and GIC4, still in the *best-case* and assuming the case of maintaining GIC (i.e., after the assignment of a variable by the user), no call to the oracle is necessary (residues and the current table permit alone to prove that all values are GIC). This rough analysis of time complexity suggests that GIC3 and GIC4 might be the best options.

5 Tuple Inverse Consistency

Up to this point, we have based our analysis on the last part of the interactive resolution process, i.e., the specification of a solution of the constraint network by the user. This allowed us to make the simplifying assumptions that the user is only looking at the domains of the variables. After each variable assignment, she just wants to know which values remain feasible for non assigned variables.

The situation is different at the modelling phase, e.g., the engineers of the company dynamically build the set of constraints that define the configurable product. At this point, GIC is also a crucial functionality, not for deriving a solution (a end product), but to ensure that each of the options proposed in the catalog (each of the values in the domains of the constraint network) is present in at least one end product. It is meaningless to propose (and advertise on) a sophisticated air bag system when it cannot equip any car in practice.

In that modelling phase, the need for information on the extensibility to solutions is not restricted to domains, but extends to (some of) the constraints of the model. Many constraints have actually a double meaning. Following the standard semantics of constraints, the first one is negative: technical constraints forbid combinations of variables. The second one is positive: the possibilities that are left by some (generally, table) constraints *have to* be effective. Let us assume, for instance, that a constraint means 'The level of equipment of vehicles with type M3 engine can be middle level or luxurious'.

If some other constraint excludes the vehicles M3 engine for luxurious level of equipment, the specification of the product is considered as inconsistent. This property has been called *positive consistency* in [2] and actually refers to the extensibility to a solution of each of the tuples allowed by the constraint of interest:

Definition 2 (Tuple Inverse Consistency). *Given a CN N, a tuple τ on a set of variable X is said to be inverse consistent (TIC) in N iff there exists a solution I of N such that $\forall x \in X, I[x] = \tau[x]$.*

Definition 3 (Positive Consistency). *A constraint c is positively consistent in N iff for any valid tuple $\tau \in rel(c)$, τ is TIC.*

The positive closure of a constraint c is the constraint obtained from c by removing from $rel(c)$ all the valid tuples that are not TIC in N. The obvious problem that follows is to check whether a constraint is positively consistent or not.

Problem 5 (Deciding Positive Consistency) *Given a CN N and a constraint c of N, is c positively consistent in N?*

Deciding positive consistency has been shown to be NP-hard, even when the constraint network is known to be satisfiable ([2]). The other problem of interest is to restore positive consistency on a constraint after the user has refined her model by adding a constraint to the network.

Problem 6 (Restoring Positive Consistency) *Given a CN N, given a positive consistent constraint c in N, given any extra constraint c' not in $cons(N)$, compute the new positive closure of c in the network obtained from N by adding c' to $cons(N)$.*

6 Experiments

In order to show the practical interest of our approach, we have performed several experiments mainly using a computer with processors Intel(R) Core(TM) i7-2820QM CPU 2.30GHz; for random instances, we used a cluster of Xeon 3.0GHz with 13GB of RAM. Our main purpose was to determine whether maintaining GIC is a viable option for configuration-like problem instances and for interactive puzzle creation, as well as to compare the relative efficiency of the four GIC algorithms described in Section 4.

	n	d	e	r	t	D	T
souffleuse	32	12	35	3	55	145	350
megane	99	42	113	10	48,721	396	194,838
master	158	324	195	12	26,911	732	183,701
small	139	16	147	8	222	340	3,044
medium	148	20	174	10	2,718	424	9,532
big	268	324	332	12	26,881	1,273	225,989

Table 1. Features of six Renault configuration instances.

In Table 1, we show relevant features of car configuration instances, generated with the help of our industrial partner Renault. For each of the six instances currently available,[4] we indicate

- the number of variables (n),
- the size of the greatest domain (d),
- the number of constraints (e),
- the greatest constraint arity (r),
- the size of the greatest table (t),
- the total number of values ($D = \sum_{x \in vars(N)} |dom(x)|$),
- and the total number of tuples ($T = \sum_{c \in cons(N)} |rel(c)|$).

The left part of Table 2 presents the CPU time required to establish GIC on the six Renault configuration instances. Clearly GIC1 is outperformed by the three other algorithms, which have here rather similar efficiency. The right part of Table 2 aims at simulating the behaviour of a configuration software user who makes the variable choices and value selections. It presents the CPU time required to maintain GIC along a single branch built by performing random variable assignments (random variable assignment simulates the user, who chooses the variables and the values according to her preference). Specifically, variables and values are randomly selected in turn, and after each assignment, GIC is systematically enforced to maintain this property. Of course, no conflict (dead-end) can occur along the branch due to the strength of GIC, which is why we use the term of greedy executions. CPU times are given on average for 100 executions (different random orderings). For all instances, GIC3 and GIC4 are maintained very fast, whereas on the biggest instances, GIC2 requires a few seconds and GIC1 around ten seconds.

	Establishing GIC with				Maintaining GIC with			
	GIC1	GIC2	GIC3	GIC4	GIC1	GIC2	GIC3	GIC4
souffleuse	0.02	0.01	0.01	0.01	0.13	0.07	**0.02**	**0.02**
megane	2.94	0.71	0.72	0.71	4.26	1.18	0.05	**0.04**
master	2.45	1.35	1.33	1.33	9.81	3.57	0.07	**0.06**
small	0.14	0.02	0.03	0.03	0.32	0.05	**0.01**	**0.01**
medium	0.26	0.04	0.05	0.04	0.35	0.04	**0.01**	**0.01**
big	4.19	1.16	1.10	1.10	12.6	2.60	**0.05**	**0.05**

Table 2. CPU time (in seconds) to establish GIC on Renault configuration instances, and to maintain it (average over 100 random greedy executions).

One great advantage of GIC is that it guarantees that a conflict can never occur during a configuration session. However, one may wonder whether the risk of failure(s) is really important in user-driven searches that use a weaker consistency such as GAC or a partial form of it (Forward Checking). Table 3 shows the number of conflicts (sum over 100 executions using random orderings) encountered when following a MAC or a

[4] see http://www.irit.fr/ Helene.Fargier/BR4CP/benches.html

nFC2 [3] strategy. The number of conflict situations can be very large with nFC2 (for two instances, we even report the impossibility of finding a solution within 10 minutes with some random orderings). For MAC, the number of failures is rather small but the risk is not null (for example, the risk is equal to 5% for megane).

	souffleuse	megane	master	small	medium	big
nFC2	252,605	313,910	time-out	3,728	7,824	time-out
MAC	0	7	5	0	3	3

Table 3. Number of conflicts encountered when running nFC2 and MAC (sum over 100 random executions).

The encouraging results obtained on Renault configuration instances led us to test other problems, in particular to get a better picture of the relative efficiency of the various GIC algorithms. For example, on classical Crossword instances (see Table 4), GIC1 is once again clearly outperformed while the three other algorithms are quite close, where there is still a a small benefit of using GIC4.

	Establishing GIC with				Maintaining GIC with			
	GIC1	GIC2	GIC3	GIC4	GIC1	GIC2	GIC3	GIC4
ogd-vg5-5	2.25	0.67	0.67	0.67	2.34	0.79	0.73	**0.70**
ogd-vg5-6	6.40	2.18	2.19	2.19	7.42	2.82	2.58	**2.48**
ogd-vg5-7	25.8	9.91	9.87	9.84	33.4	15.2	14.3	**13.8**

Table 4. CPU time (in seconds) to establish GIC on some Crosswords instances, and to maintain it on average over 100 random greedy executions.

	Establishing GIC with				Maintaining GIC with			
	GIC1	GIC2	GIC3	GIC4	GIC1	GIC2	GIC3	GIC4
sudoku-9x9	1.58	0.32	0.32	0.31	15.3	2.71	2.10	**1.74**
sudoku-16x16	6.04	0.51	0.50	0.50	246	25.5	26.5	**18.9**
magicSquare-4x4	0.96	0.26	0.28	0.28	1.63	**0.69**	0.71	0.71
magicSquare-5x5	14.7	3.01	3.10	2.99	55.1	15.9	15.6	**13.7**

Table 5. CPU time (in seconds) to establish GIC on Puzzle instances, and to maintain it on average over 100 random greedy executions until a unique solution is found.

It is worthwhile to note that GIC is a nice property that can be useful when puzzles, where hints are specified, have to be conceived. Typically, one looks for puzzles where only one solution exists. One way of building such puzzles is to add hints in sequence, while maintaining GIC, until all domains become singleton. For example, this is a possible approach for constructing Sudoku and Magic Square grids, with the advantage that the user can choose freely the position of the hints.[5] On the left part of Table 5,

[5] However, we are not claiming that maintaining GIC is the unique answer to this problem.

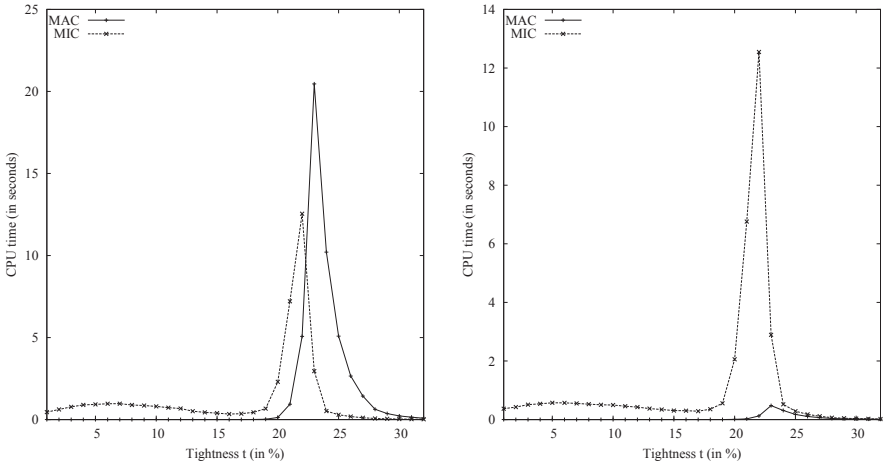
we report the time to enforce GIC on empty Sudoku grids of size 9x9 and 16x16, and on empty Magic square of size 4x4 and 5x5, and on the right part, the average time required to maintain GIC until a fixed point is reached, meaning that after several hints have been randomly selected and propagated, we have the guarantee of having a one-solution puzzle. GIC4 is a clear winner, with for example, a 30% speedup over GIC2 and GIC3 on sudoku-16x16, and more than one order of magnitude over GIC1. Overall, the results we obtain show that MIC, i.e., maintaining GIC, is a practicable solution (at least for some problems) as the average time between each decision of the user is small with GIC4.

The efficiency of MIC on structured under-constrained instances piqued our curiosity. So we decided to compare MIC (embedding GIC4) and MAC on series of binary random instances generated from Model RB [20]. For the class $RB(2, 30, 0.8, 3, t)$, see [19], we obtain instances with 30 variables, 15 values per domain and 306 binary constraints of tightness t, and for the class $RB(2, 40, 0.8, 3, t)$, instances with 40 variables, 19 values per domain and 443 binary constraints of tightness t. For each value of t ranging from 0.01 to 0.50 (step of 0.01), a series of 100 instances was generated so as to observe the behaviour of MIC on both under-constrained instances and over-constrained instances; the theoretical threshold is around 0.23. Figure 1 shows the average CPU time of MIC and MAC on series of class $RB(2, 30, 0.8, 3, t)$. On the left, Figure 1(a), the ordering of variables and values is random (simulating a free user-driven search). MIC outperforms MAC when the ordering is random and the tightness is greater than or equal to 0.23. That means that the strong inference capability of MIC do pay off for the unsatisfiable instances. On the right, Figure 1(b), the variable ordering heuristic is *dom/wdeg* [4] and the value ordering heuristic is lexico. Obviously, MAC with *dom/wdeg* is clearly faster than MIC. However, if used in a context of interactive resolution, the *dom/wdeg* ranking of the variables drives the user, who is not free anymore in the choices of its variables. It may ask her to assign first variables that are meaningless to her, restricting her future choices on important variables. The outcome will be a solution which is very bad with respect to the preferences of the user. All of this suggests that MIC can be efficient enough to be used in practice, except for a (small) region of satisfiable instances lying at the left of the threshold point. Figure 2 shows similar results with respect to series of class $RB(2, 40, 0.8, 3, t)$.

One other practical issue we are interested in is the effectiveness of positive consistency. Hence, we tested to establish positive consistency on existing constraints of the Renault configuration instances, see Table 6. The algorithm we used here is a simple adaptation of GIC1 to tuples (so, certainly, several optimizations are possible). A few hundreds of seconds are necessary to ensure the positive consistency of all existing constraints of the biggest instances.

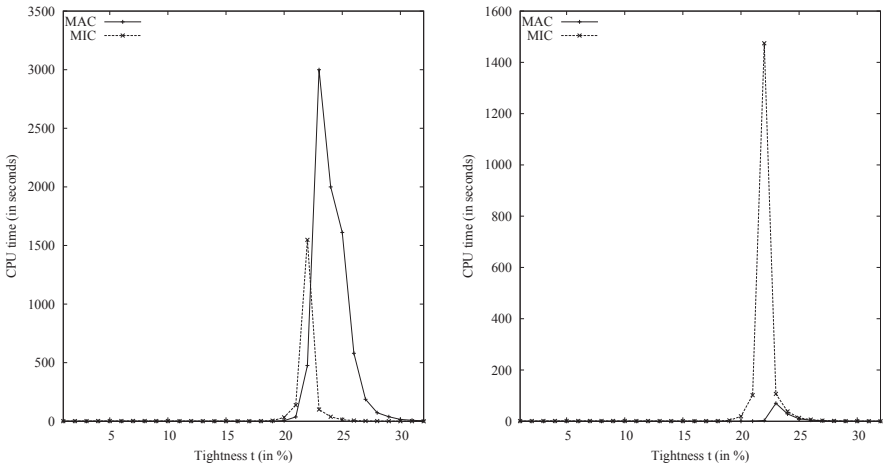
	souffleuse	megane	master	small	medium	big
CPU	0.68	352	368	2.6	4.2	613
# tuples removed	0	138,493	90,874	240	5,425	105,020

Table 6. CPU time (in seconds) and filtering in term of the number of tuples deleted when establishing positive consistency on Renault configuration instances.



(a) With random ordering (b) With heuristic *dom/wdeg*

Fig. 1. Mean search cost (100 instances) of solving instances in class $RB(2, 30, 0.8, 3, t)$ with MAC and MIC.



(a) With random ordering (b) With heuristic *dom/wdeg*

Fig. 2. Mean search cost (100 instances) of solving instances in class $RB(2, 40, 0.8, 3, t)$ with MAC and MIC.

Finally, in our last experiment, for each constraint network, we randomly select a constraint of interest c_i for which positive consistency must be ensured (as if the modeller were asking for the positive consistency of this constraint), and we randomly select a set C containing 10% of the set of constraints. We initially consider the CN without the constraints in C, and we enforce positive consistency on c_i. Then we simulate a session of product modelling: we post each constraint in C in turn and maintain positive consistency on c_i. In our implementation (not detailed here due to lack of space), we use residues, i.e., a solution stored for each tuple of c_i. The first line of Table 7 shows the average CPU time to maintain positive consistency on the constraint of interest. For the second line, the constraint of interest is not randomly chosen but set to the constraint with the largest table. The obtained results are rather promising (except for the instance megane).

	megane	master	big
random	9.97	10.1	36.4
largest	106.6	11.4	20.6

Table 7. Dynamic positive consistency filtering on Renault configuration instances (average CPU time over 100 executions).

7 Conclusion

We have analysed the problems that arise in applications that require the interactive resolution of a constraint problem by a human user. The central notion is global inverse consistency of the network because it ensures that the person who interactively solves the problem is not given the choice to select values that do not lead to solutions. We have shown that deciding, computing, or restoring global inverse consistency, and other related problems are all NP-hard. We have proposed several algorithms for enforcing global inverse consistency and we have shown that the best version is efficient enough to be used in an interactive setting on several configuration and design problems. This is a great advantage compared to existing techniques usually used in configurators. As opposed to techniques maintaining arc consistency, our algorithms give an exact picture of the values remaining feasible. As opposed to compiling offline the problem as a multi-valued decision diagram, our algorithms can deal with constraint networks that change over time (e.g., an extra non-unary constraint posted by a customer who does not want to buy a car with more than 100,000 miles except if it is a Volvo). We have finally extended our contribution to the inverse consistency of tuples, which is useful at the modelling phase of configuration problems.

One direct perspective of this work is to try computing diverse solutions when enforcing GIC. This should permit, on average, to reduce the number of search runs. Some techniques developed in [11] might be useful.

References

1. J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs - application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.
2. J.M. Astesana, L. Cosserat, and H. Fargier. Constraint-based vehicle configuration: A case study. In *Proceedings of ICTAI'10*, pages 68–75, 2010.
3. C. Bessiere, P. Meseguer, E.C. Freuder, and J. Larrosa. On Forward Checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141:205–224, 2002.
4. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
5. R. Debruyne and C. Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
6. E.C. Freuder and C.D. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI'96*, pages 202–208, Portland, Oregon, 1996.
7. E. Gelle and R. Weigel. Interactive configuration using constraint satisfaction techniques. In *Proceedings of PACT'96*, pages 37–44, 1996.
8. G. Gottlob. On minimal constraint networks. *Artificial Intelligence*, 191-192:42–60, 2012.
9. T. Hadzic and H.R. Andersen. Interactive reconfiguration in power supply restoration. In *Proceedings of CP'05*, pages 767–771, 2005.
10. T. Hadzic, E.R. Hansen, and B. O'Sullivan. Layer compression in decision diagrams. In *Proceedings of ICTAI'08*, pages 19–26, 2008.
11. E. Hebrard, B. Hnich, B. O'Sullivan, and T. Walsh. Finding diverse and similar solutions in constraint programming. In *Proceedings of AAAI'05*, pages 372–377, 2005.
12. P. Janssen, P. Jégou, B. Nouguier, M.C. Vilarem, and B. Castro. SYNTHIA: Assisted design of peptide synthesis plans. *New Journal of Chemistry*, 14(12):969–976, 1990.
13. C. Lecoutre. STR2: Optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.
14. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.
15. D. Martinez. *Résolution interactive de problemes de satisfaction de contraintes*. PhD thesis, Supaero, Toulouse, France, 1998.
16. C. Papadimitriou. private communication, 1999.
17. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.
18. J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.
19. K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8-9):514–534, 2007.
20. K. Xu and W. Li. Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 12:93–103, 2000.